# UPDATE — A Stealthy Sniffer Detector
# Part I

What was a security-minded system administrator to do? The requirement for installation of sniffer detector software on all Unix machines came down from 'on-high,' but how could the sniffer detector be hidden, so that potential crackers couldn't find it? The search for an answer to that question led the author on an odyssey of discovery and invention, and led to the development and deployment of the update sniffer detector.

In this two-part article, we will examine the design philosophy behind the implementation of update, the technical challenges overcome, innovative features added, and practical experience gained from the use of update.

## WHAT IS A SNIFFER DETECTOR AND WHY IS IT IMPORTANT TO INSTALL ONE?

Sniffer detectors are a standard part of a security toolkit to protect computing assets from hostile attack. Once a cracker gains access to a Unix machine on a network, one common action they will take is to install sniffer software that passively listens on the network for interesting traffic, particularly passwords to other systems, financial or other proprietary data, and personnel data. This information can allow them to penetrate further into your network. A sniffer detector will alert the system administrator if the network card is set to sniff network traffic.

It is important to recognize that the installation of sniffer detector software is a second-tier defense. If the sniffer detector fires off, this means that your network has already been penetrated. By receiving and responding to a sniffer detector alert, the intrusion can be limited in scope, and halted before further serious damage is incurred.

Network interfaces on systems are generally set to respond only to traffic addressed specifically to that system, or network broadcast traffic. Rather than burdening the CPU with checking and discarding much of the network traffic, this burden is shifted to the network card. In the vast majority of cases, this is the standard mode of operating.

For specialized applications, such as network monitoring (tcpdump, snoop, etc.), network cards allow another mode of operation (called 'promiscuous mode') which transfers all packets received to the CPU for further processing. Those applications which process all packets can arrange to receive them all, whereas other applications on the same system must receive their normal expected traffic. This additional processing entails an extra burden on the CPU, so it is not the normal mode of operation. The kernel keeps a flag indicating whether this extra processing is needed.

Promiscuous mode is also used by sniffers. A sniffer detector checks whether the network interface is in promiscuous mode, which is almost never appropriate for a general purpose machine. This is easily done in BSD-derived systems via a documented interface. As we shall see, some other platforms present unique challenges.

When a sniffer is detected, what is the appropriate response? Certainly, the system administrator should be notified. As sys admins are individualistic (often to the extreme), the response needs flexible enough to satisfy their needs.

## SNIFFER DETECTORS IN GENERAL USE

Several sniffer detectors are being generally used and are freely available:

- cpm (from Purdue)
- ifstatus (David Curry, IBM)

## THE STORY

While working as a Unix System Administrator at a large organization, management decreed, in response to security concerns, that all Unix systems have sniffer detector software installed. Naturally, the decree was painted with a broad brush, and implementation details were left to the individual administrators.

The obvious default path to take was recommended in the documentation for the available sniffer detectors: Install a periodic cron job that would check for the promiscuous flag being set, and email the administrator if promiscuous mode was set, something like this:

```
0,5,10,15,20,25,30,35,40,45,50,55  *  *  *  *  /uar/local/bin/cpm ||
echo Promiscuous mode detected | mail root
```

This works because cpm returns an exit status which is the number of interfaces detected in promiscuous mode.

This is simple, functional, and easy for a cracker to discover and disable. Slightly better would be to put the sniffer detector checking in an innocuous sounding shell script, invoked periodically via cron. Another possibility would be to invoke it in a sub-shell at bootup time, say in /etc/rc2.d/S99local:

```
while:
do
     /usr/local/bin/cpm || echo Promiscuous mode detected | mail root
     sleep 300     # 5 minutes
done &
```

This is perhaps somewhat more elegant than cron, since it is not quite the documented/ recommended configuration. Another possibility could be to take advantage of the ‘respawn’ capability of /etc/init to periodically execute the sniffer detector, via an entry in /etc/inittab.

These methods certainly would work, but I figured that if I could think of them, then a wily cracker could too, and perhaps take revenge on the system administrator who attempted to fool him/her.

Wouldn't it be nice to run a legitimate system service with the sniffer detection grafted in, stealthily?

## ENTER UPDATE

Since its earliest days, Unix, like most other modern operating systems, was designed to defer writing data to disk for as long as possible or practical. Instead, the data is cached in memory for later output when the system is idle. The efficiency gains can be enormous, especially in typical data processing tasks, such as file copying or conversion. Rather than reading a block from the source file, moving the disk head, then writing a block to the destination file, this deferred output scheme allows reading multiple sectors, and only moving the head and writing data when the buffer became full.

It was recognized that it was desirable to not allow deferring of writes indefinitely, however. Operating system crashes or potential power failures dictate that data be physically written to the disk as soon as possible. Efficiency pales in importance if data can potentially be lost.

It can be seen that these two factors are somewhat incompatible, so the designers of Unix decided on a compromise: On a regular, periodic basis, all disk buffers destined to be written to disk will be physically dumped out. Empirically, it was determined that a period of 30 seconds was an adequate compromise. This ensured that no data would wait more than 30 seconds (disregarding disk latency, or other minor delays) to be written out.

Rather than build this strategy into the kernel, early Unix implementations used a daemon which makes a sync() system call every 30 seconds. sync() schedules unwritten buffers to be written as soon as possible. This simple daemon can be written in a few lines of C code:

```
main()        {
        while (1)    {
              sync();
              sleep(30);
        }
}
```

Modern Unix implementations use a much more sophisticated buffer flushing strategy to avoid the periodic processing stalls that this simple scheme can entail if large amounts of data are dumped to the disk at once.

Hmm, I wonder if the sniffer detector function can be added to the simple buffer flushing scheme above? This is an ideal candidate, since it is a small footprint, periodically run system daemon that most crackers wouldn't give a second thought to.

## INITIAL STEPS

Suppose we wanted to check the promiscuous flag every five minutes. We could count ten intervals of 30 seconds then check promiscuous mode, like this:

```
main()        {
          int i=0;
          while (1)        {
                sync();
                sleep(30);
                if ( ++i == 10 )         {
                i=0;
                if (promiscuous_mode())
                    system("echo Promiscuous mode found | mail root");
    }
    }
}
```

We assume promiscuous_mode() is a function returning non-zero if any interface is in promiscuous mode. This, in fact is essentially what was done in the first version of my revised update. After surveying applicable tools, I chose to use the engine from ifstatus, since it supports a larger range of systems than cpm.

This is clearly functional, and a drop-in for the original update. On those systems which natively use the update daemon, this should drop in, with few modifications. There are a few issues, though:

1. The original update program automatically daemonizes itself. This is easily accomplished by a few lines of code to fork() a child process, then exit the parent. To be absolutely correct, a call to setsid() is needed, as well, to create a new process group.

2. For the sake of robustness, the original function of the update daemon must be protected, regardless of possible problems with the promiscuous mode checking, or shell script executed by the system() function. In the example code above, if the shell script should stall, the sync() call can be indefinitely delayed, thus preventing disk buffers from getting flushed. The easiest fix to this is to execute the promiscuous mode checking and system() function in a child process, again by fork()ing. As we shall see, even this precaution has its problems.

With these minor modifications, I felt confident enough to release update 0.9BETA to my peers in the community.

## REACTION/LESSONS LEARNED

System Administrators embraced the program, end of story. Right!

Although it proved very useful to administrators, there were a few comments and ideas which prompted further development:

1. Administrators didn't like to recompile the program to customize 'update 0.9BETA' for their particular platform. Furthermore, although C compilers are freely available, some admins simply didn't have the time to download and install 'gcc' simply to install update.

2. Running 'strings' on the update binary printed out the full shell script to be executed when promiscuous mode detected — a little too revealing for some admins.

3. The most serious problems were with the promiscuous mode engine (ifstatus). Although a useful program, it failed on some of the platforms deployed on site:

    - 32-bit Solaris on the hme interface was not reliable, and no support was provided for some fiber cards.
    - 64-bit Solaris (UltraSparc processors) simply didn't work at all.
    - Solaris x86 didn't work.
    - Linux 2.2.x kernel didn't work, although the older 2.0 kernel does work with the ifstatus engine.
    - AIX didn't seem to be able to determine the network interfaces in use.

Of these 3 difficulties, problem #3 seemed the most severe, and also the most challenging, since up to this point, I had been leveraging off the work of others. To solve these difficulties would require hacking the kernel, without access to source code (at the time true for Solaris). However, not to disappoint the community, I decided to tackle these problems.

## DETECTING PROMISCUOUS MODE ON BSD-DERIVED UNIX

Before embarking on uncharted waters, it is worthwhile to consider the easy and reliable promiscuous mode detection available on BSD-derived systems. The BSD socket interface is simple and reliable and thus has been adapted by many platforms. Even on those Unix variants that use Streams-based networking, emulation libraries allow transparent access to most features using socket calls.

Once a socket is opened, an ioctl() call (SIOCGIFCONF) can be made to fill a buffer with an array of structures containing per-interface information. By stepping through this table, the interface name is extracted and the flags determined via another ioctl() call (SIOCGIFFLAGS). If the IFF_PROMISC bit is set, the interface is in promiscuous mode. This scheme requires no poking around in the kernel, rather uses the documented interface. Those systems using this paradigm required no changes in the ifstatus engine to detect promiscuous mode. However, not all systems are so compliant, most notably Solaris.

## DETECTING PROMISCUOUS MODE ON SOLARIS

Inspecting the ifstatus source code for Solaris began the process of discovery. Ifstatus looks in the kernel namelist for the structure containing the interface flags. The correct word is extracted

from kernel space and checked for promiscuous mode. This strategy seems correct and foolproof — what could be wrong?

Poking around in the kernel with adb, while running 'snoop' in another window revealed the plain truth — on the hme interface, at least, the interface flags are unreliable. In some cases they worked, but in other circumstances (the difference not being obvious) the flag was not set when a program entered promiscuous mode. Clearly, the kernel must know that the interface is in promiscuous mode to deliver the packets to 'snoop,' but finding the key to the problem was the hard part.

Finally, a fellow sysadmin told me of a conversation with a Sun kernel engineer. The engineer said that in order to detect promiscuous mode on the hme interface, each individual stream needed to be checked. This statement opened my eyes: I had seen the 'per-stream' information in hme.h, along with flags that indicated promiscuous mode attached to the stream. This must be the Mother Lode! Now to figure out how to mine it.

The header file shows that the 'per-stream' information is kept on a linked list. My hope was that a kernel variable would point to the beginning of this linked list. I printed out the kernel symbols with 'adb,' then grep'ed through them for hme. hmm, the structure is named 'hmestr' in the header file, perhaps this variable 'hmestrup' points to it. Poking about with adb appeared to show a linked list, with the last link being a null pointer in the 'next-link' field — just what would be expected! Feeling elated, I took a break, and went outside to smoke a cigar.

To verify my assumption, I added a compile mode which skipped the daemonization of 'update' and added printf() statements to print out data structures along the way. This proved to be invaluable in determining whether this kernel variable indeed pointed to the expected structure. Experimentation proved that it was the correct kernel variable . . .

After this success, I decided to check if the other Sun adapters followed the same strategy — I was uncomfortable with the unreliability (in at least the case of hme) of the interface flags. The header files revealed that the other adapters also had 'per-stream' information, with somewhat different names, but the same basic scheme held true, so all the other known interfaces were compiled & tested with this schema.

So, at this point, I had a functioning sniffer detector for Solaris, which was an advance on the state of the art. Then Sun unleashed Solaris 7, which by default runs a 64-bit kernel on UltraSparc platforms. The first indication that this might be a problem was a call from another sysadmin complaining that update didn't work on his new Ultra 5. Since it worked fine on my IPX, I originally attributed this to 'cockpit error.' As reports piled in, I could no longer ignore the 64-bit Solaris world.

## A SINGLE BINARY FOR 32-BIT & 64-BIT

Once I got hold of a shiny new Ultra 5 myself, I could no longer hold off on providing 64-bit support. One self-imposed criterion was that the same binary should run on both 32 & 64 bit

kernels. After all, you can boot UltraSparcs with a 32-bit kernel by the incantation: 'boot kernel/unix,' and one of the loadable modules I needed for my day job was 32-bit only. Rather than supply two binaries, I wanted a single update binary that would autodetect the current running kernel and branch to the correct code tree.

The essential problem, verified with adb, was that pointer variables were now 64-bit, so that the structure offsets were different. The data was still accessible via the same interface(s), if the offsets were adjusted.

Since I wanted a single binary for both platforms, it had to be a 32-bit application, since only 32-bit apps run on both platforms. A major problem was that the kernel name list functions in a 32-bit application could not access the 64-bit kernel — this was understandable, due to the difference in pointer size. This loomed as a potential show-stopper — I feared that I would have to write my own ELF access routines to solve this. I downloaded the ELF specifications, and found the complexity discouraging. Briefly, I considered having update fork off a shell to adb, printing out & parsing the namelist, although this would be a terrible hack that violated my sense of esthetics.

After playing with adb for a while, I discovered a ray of hope — although the kernel space is 64-bit, the kernel name space resides in the first 32-bits. What that meant was, except for the restriction imposed by the Sun libraries, I should be able to capture the beginning of the 'linked-list' (remember that?) in a 32-bit pointer. I downloaded & compiled a GPL'd libelf source tree, linked that to update, and was able to access the linked list. It turned out that subsequent entries in the list do reside in the full 64-bit space, but at least I had a starting point.

The source code also gave the answer to easy determination of kernel type. It turns out that the 5th byte of an ELF file (in this case /dev/ksyms) contains the ELF class, 1 if 32-bit, 2 if 64-bit. This file is dynamically based on the running kernel, so determining the kernel was only a few lines of code. Easy, once you figure it out . . . .

At this point, what was needed was to modify the header files to allow 64-bit pointers. Since accessing the kernel data space is done via lseeks and reads of /dev/kmem, a pointer type wasn't necessary, only a 64-bit type, which is supplied with the long long type. Also, duplicate code was developed using these 64-bit headers & types, and branched to only if the kernel was determined to be 64-bit. Once the roadmap became clear, this work flied — finding the path was the hard part.

## SOLARIS X86

An admin wanted the software ported to Solaris X86 and offered up a machine for use. Presumably, the structure would be similar, I hoped. Looking thru the header files, I discovered that Solaris X86 had been simplified — rather than have a separate header file and structure for each possible network card, Sun created an abstraction layer above the hardware details of each card. They created a linked list of interface structures, each of which has a pointer to the 'per-stream' linked list that I had come to know and love.

Displaying the data in debug mode, I discovered Sun was using the RAW flag in a stream to indicate promiscuous mode. Other than that, there seemed to be no surprises, so I coded them detection module, and started it running right before going home for the evening, on the user's machine. Little did I know that this was a production machine!

The next day the admin called and reported that the machine was running at a snail's pace & there were hundred of processes named update running. I logged on and killed the jobs, then set out to investigate the problem. As discussed earlier, the detection module is actually run as a child process to the main program, so each time the sniffer detector fired off, another process was spawned. These processes should have checked promiscuous mode, then immediately exited, but many of the child processes had consumed a considerable amount of CPU time, so there was clearly a problem with update getting into an infinite loop.

Investigation revealed that the linked list in Solaris X86 is organized as a ring, rather than terminated with a null pointer, so update was continuously cycling thru the list. This turned out to be the case in both the list of interfaces, and the per-stream list. The fix is to save the first address and break out of the loop when that address was seen again.

As a safety measure, a watchdog counter was also added to the loop: In case the loop goes completely bonkers, the watchdog counter causes the loop to terminate unconditionally after an empirically determined number of items were examined. This feature adds assurance that, even in the event of some catastrophic failure of the code, it will terminate rapidly and not cause the denial of service experienced by my guinea-pig administrator, who was much happier with me when I resolved the issue. The operative principle has been to keep update running lean and mean, while avoiding the many pitfalls on the way. As an insurance measure, the watchdog code was also added to the Solaris code.

-------- END OF PART 1 -------

## REFERENCES:


Sniffing (network wiretap, sniffer FAQ):
http://www.robertgraham.com/pubs/sniffing-faq.html

CPM (Check Promiscuous Mode):
ftp://ftp.cerias.purdue.edu/pub/tools/unix/sysutils/cpm

AntiSniff: http://www.l0pht.com/antisniff

Ifstatus: ftp://coast.cs.purdue.edu/pub/tools/unix/sysutils/ifstatus

Detecting Sniffers on your Network:
http://www.securiteam.com/unixfocus/Detecting_sniffers_on_your_network.html

# UPDATE — A Stealthy Sniffer Detector: Part II

## Jim Mellander

### Introduction

Part 1 (in last month's *Information Security Bulletin*) of this two-part paper series discussed some of the issues surrounding the use of sniffer detectors. The previous part covered why they are needed, the current crop of sniffer detectors in widespread use, what motivated developing a new stealthy sniffer detector, and development problems and solutions. Part 2, the current and final part of this series, discusses customizing the response that UPDATE provides, adding encryption for a stealthier "footprint," supporting a wider range of platforms, installing and using this tool (including experiences that occurred from running it), and, finally, how to obtain a copy.

### Customized Response

Recall from Part I that although UPDATE worked and delivered the basic functionality that was originally envisioned for it, it originally also had a few limitations. To address these issues required forging ahead on several fronts: porting continued to expand the range of Unix platforms supported, and several items on the "complaint list" were examined and addressed. Administrators wanted an easy way to customize the shell script without using a compiler; they were also concerned that the text of the shell scripts was visible in the binary version. These "complaint list" items are actually related, as we will see shortly.

The solution that I chose to correct these complaints was to set aside space in the binary with a recognizable signature, then actually patch the binary with the desired shell script. A large string was set aside as the command to be run by the system() function, filled with repeated instances of "PATCH_ME_HERE_UPDATE." A patch program was developed which looks for that signature in the binary, checks that an excessively large shell script will not overflow the buffer, and replaces the string data with the shell script.

I developed a script that standardized the most common reporting actions that I anticipated a system administrator would want. Any of the following options could be selected as desired:

- Report via email
- Report via pager
- Report via the syslog facility
- Turn off network interfaces
- Shutdown system

Each of these options generates a shell script fragment. Shell script fragments are in turn combined to create a script to implement these actions. The first three options, email address, pager PIN, and syslog logging level, prompt for further information. The last two options contain a built-in delay, so that the previous options have a chance to function *before* network

connectivity is dropped or the system is shutdown (which also, of course, also results in dropping network connectivity).

The shell script is displayed for the administrator, who can edit it to select customized actions. Because the default script has to work across multiple platforms, it includes a search path sufficient to cover all the supported platforms. The script uses only standard installed Unix commands. For instance, the shell command to turn off the network interfaces is implemented portably by entering:

```
sleep 5;netstat –rn | awk '$NF !~ /^lo*/ {print "ifconfig " $NF "
down&”}’|sh
```

When this command is executed, the interface names from the routing table are extracted and ifconfig `<NAME OF INTERFACE> down`[1] is executed on all interfaces except the loopback. Standard output and standard errors are redirected to `/dev/null` before the shell script is run, preventing error messages from being printed.[2] When the script is finalized, the binary is patched with the script. Once this operation is done once, the binary can be deployed on all similar systems in an organization with the same reporting action policy.

## A Dilemma: How to Protect Security Measures

Running `strings` on the UPDATE binary displays the shell script being executed, a dead giveaway when the email message “Alert — Promiscuous Mode Detected” is embedded in the script. Although it seems unlikely that an attacker would scan the binary, the threat of someone doing this seemed troubling. For better or worse, UPDATE uses the “security by obscurity” paradigm by hiding security measures so that attackers are unaware of their presence.[3] “Security by obscurity” is not always bad; in this case we have in effect at least added another barrier that an attacker must address to avoid detection. By obfuscating the command to be executed, we create more potential work for an attacker and also possibly defeat casual attacks.

Encryption of the shell script provides a solution. If the script is encrypted when it is embedded in the binary, then decrypted when executed, we can at least accomplish the goal of hiding the script from the `strings` command.

Many well-known, cryptographically strong algorithms with easy-to-use APIs could be used for this purpose. These, however, exceed the needs of UPDATE for several reasons:

---

[1] This is a command that shuts the interface (e.g., eth0 in Linux) down.

[2] These messages would otherwise be printed due to the heading printed by the netstat command in some platforms.

[3] In a sense, publicizing the UPDATE program in this paper defeats this paradigm, since attackers may now become aware of this software.

1. These packages are designed for secure communication of data, rather than simple obfuscation. As such, they provide more functionality than needed. They also expand the code size considerably.

2. Once the presence of UPDATE is detected by an attacker, the game is over. An attacker with root privileges can easily kill the process to stop the sniffing activity. The actual shell script executed when promiscuous mode is detected is in this sense irrelevant to an attacker, who can at least be confident that it notifies security personnel. The fact that a sniffer detector is running is thus in reality the critical information being protected. This information is fundamentally different from the type that cryptographic algorithms protect.

## Simple Encryption with XOR

With these caveats in mind, I decided on a simpler, tinier, encryption scheme. UPDATE's cryptographic capabilities depend on the properties of the Boolean function XOR. If a value is XOR'd with any key (except 0), a different value is returned. If this returned value is again XOR'd with the same key, the original value is retuned. In our application, the shell script is encrypted by XOR'ing each byte with a reproducible pseudorandom byte stream. To decrypt, the encrypted (using the term loosely) stream is again XOR'd with the same pseudorandom byte sequence. C provides the rand() function to generate random integers, as well as the srand() function to seed the rand() function with a known starting point. This seemed ideal for this encryption scheme.

Naturally, things become more complicated. Vendors have been known to change library functions such as rand() and srand() to improve randomness and to add functionality.[4] I became concerned that a binary built on an older version of an operating system may fail (silently!) when an upgrade occurs. To address these concerns, simple implementations of rand() and srand() are included in UPDATE.

Another additional detail: If the shell script does not completely fit into the space allocated, the "signature" of the placeholder (PATCH_ME_HERE_UPDATE) could be revealed. Thus, the encryption phase encrypts all the way to the end of the placeholder. Decryption, however, stops when the first zero byte is decrypted signaling the end of the embedded shell script, since there is no need to continue.

## Supporting More Platforms

AIX uses the BSD paradigm (with a "wrinkle"), while Linux is an evolving operating system, with differences primarily based in the kernel. We will start with AIX.

[4] Frankly, the feared behavior by vendors has not generally manifested itself in practice. During development of critical security software, nevertheless, heeding paranold inclinations should be considered desirable.

### AIX

As with BSD-Unix, an ioctl() call (SIOCGIFFLAGS) retrieves the flags for a known interface, and the IFF_PROMISC bit reliably indicates the state of the interface. Readers will remember that the list of installed interfaces is derived by executing the SIOCIFCONF ioctl(), which fills a buffer with a list of the interfaces installed on the system. Unfortunately, on AIX, this buffer came back filled with garbage, rather than useful information. I was unable to determine the cause of this phenomenon, although, of course, it is quite possible that it was a "short-circuit between the ears." AIX being a minor platform in our installation, I decided to "punt": the AIX code simply has a hard-coded list of standard interfaces to step through. Although a kludged scheme, in practice it has proven successful. For systems with additional interfaces, the code will need to be modified to accommodate the additional interfaces. For reference, the interfaces checked by UPDATE on AIX are: en0, en1, ent0, ent1, et0, et1, fi0, and fi1. Aside from this wrinkle, the standard BSD code is used. UPDATE has been successfully compiled and used on AIX 3.5.x and AIX 4.2.x.

### Linux

The popularity of Linux, a freely available Unix implementation with a large software base, motivated me to port UPDATE to this operating system. UPDATE has been successfully compiled and used on the Linux 2.0 kernel, the Linux 2.2 kernel, and Sparc Linox (2.2 kernel). The next section of this paper discusses the Linux implementation of UPDATE.

### Special Considerations with Linux

Although it uses a BSD-style socket interface, Linux has an independently developed networking implementation. Up to the 2.0 series kernels, the BSD-style code for UPDATE worked perfectly. When the 2.2 kernel came out, I began receiving complaints that UPDATE was not working. After verifying that this was the case, I began solving this problem. One advantage of Linux is the readily available source code for all aspects of the system, allowing inspection of the internals of the networking code.

I found that the Linux 2.2 kernel has two sets of flags, the standard flags for the interface, and a set of "global" flags that contain the "global" promiscuous and multicast bits. For some reason that my research failed to uncover, when the SIOCGIFFLAGS ioctl() is made, the actual promiscuous and multicast bits of the interface (which are correct!) are overlaid with the "global" bits. The purpose of those "global" flags remains a mystery.

My initial solution (which proved unsatisfactory) was to simply modify the kernel code to actually return the standard flags in response to the SIOCGIFFLAGS ioctl(). This worked fine with no apparent difficulties. However, I was uncomfortable with the notion of requiring users to patch and rebuild the kernel to use UPDATE. I wanted to be able to use a standard kernel distribution.

Fortunately, the source code provides the solution. The kernel variable `dev_base` points to a linked list of the network interfaces installed on the system. Stepping through this linked list and reading the structure data for each interface allows the standard flags to be accessed. This linked list always starts with the loopback interface, and is terminated with a null pointer.

When first programming the code, I did not understand that the kernel symbols for the currently running kernel are easily accessible via the `/proc/ksyms` file. Instead, UPDATE looks for the standard locations of the System.map file, then reads and parses the file for the `dev_base` symbol. To protect against the System.map file being old or obsolete, the pointer is checked for sanity before use. Once I became aware of `/proc/ksyms`, I realized that the sanity checking is unnecessary, but I left it in place anyway because it was already there and did not hurt anything. It also provided a double check that the code was correct. Once `dev_base` is determined, the first item on the list is skipped, since it is always the dummy loopback interface, then the other interfaces are checked until promiscuous mode is found or a null pointer is encountered.

### Vmware: A Wrinkle

Vmware is a popular Linux application. Through virtualization techniques, vmware allows Windows operating systems to run unmodified as "guest" operating systems under Linux. To run unmodified Windows networking applications while still supporting Linux, vmware creates a "virtual" networking interface with a different address from that used by Linux. In order to implement this, vmware puts the interface into promiscuous mode. This will cause UPDATE to fire off a false alarm. Unfortunately, turning off promiscuous mode causes vmware networking to fail. Thus, the only solution seems to be to ignore the promiscuous mode flag while vmware is running. Unfortunately, this creates an opportunity for an attacker to "slide under the UPDATE radar" if an attacker is aware of this fact.

Rather than coding special cases such as this in the base code, I decided to incorporate the vmware detection code in the shell script that is executed. The ps command is executed, and grepped for vmware. If the string is detected, the shell script immediately exits without taking any action. Although a slight inefficiency is introduced by performing this recurring action which merely exits, it seemed to be the easiest method for dealing with this issue. It also provided a model for other applications that may exhibit the same behavior.

### Other Platforms

AIX and Linux are not the only platforms to which UPDATE has been ported. Other platforms to which UPDATE has been ported include IRIX 6.5.x, Solaris 2.5 and above, Sun OS 4, Digital Unix (now Tru64 Unix), HP/UX 10, FreeBSD and others. All of the platforms not specifically described in this two-part series use the BSD style networking code, and required no additional modifications.

## Disabling Promiscuous Mode

One frequently requested option was the ability to turn off promiscuous mode on interfaces on which it is enabled. On some systems (those for which promiscuous mode detection is not difficult!), it was easy to turn it off using the documented SIOCSIFFLAGS call to set the promiscuous bit off. On the more difficult systems, turning off promiscuous mode is as difficult as discovering it in the first place! In Solaris, for instance, the per-stream information must be rewritten back to the kernel to turn off promiscuous mode for that stream. I was unable to get this feature to work in other platforms (Solaris X86, Linox 2.2); in some cases, networking instability resulted. This feature is thus disabled in these platforms.

Since the option to disable promiscuous mode is performed in the UPDATE binary, it is not coded as a response via a shell-script. It is instead hard-coded into UPDATE for those platforms on which it works. When it works, the results are quite dramatic — running `snoop` as root on Solaris will cause all network traffic for the box to start scrolling on the display terminal. When UPDATE turns off promiscuous mode, `snoop` keeps working, but only local (and broadcast traffic) are displayed. Unless an attacker is familiar with normal network traffic generated by a system, it is quite possible that an attacker may conclude that there is nothing of interest on the network, anyway. By then, the alert system administrator will be aware of the sniffing activity.

## Installing and Testing UPDATE

The next section of this paper covers how to install and test UPDATE. First, I'll describe how to install this tool.

### Installation

Installing UPDATE requires several steps. First, download the distribution from *ftp://lassie.lbl.gov/UPDATE/,* then `gunzip` and `untar` the distribution. Pending licensing issues, only a binary distribution is provided. To install the precompiled binaries, go to the `dist` sub-directory, and copy the `UPDATE.in.*` file for your platform to `UPDATE.in.` Then copy the `patcher.*` file for your distribution to `patcher`. Patch the binary by first entering:

```
./patch.sh
```

The five previously discussed reporting methods (e.g., email, pager, and so forth) will now be displayed. You can select any combination of these five. Enter the number of the option you desire, then press `<Enter>`. You can enter multiple options separated by any characters (or none at all) on one line. For instance, entering "12345" causes all five options to be selected, as will "1,2,3,4,5." The menu will be redisplayed with an asterisk (*) next to each selected option. To deselect an option, simply enter the option number again. When finished, press 0 to continue with the configuration. Depending on the options selected, you must now enter additional information:

- For the email configuration, you will be prompted for the email address.

- For pager configuration, enter the PIN of the pager to "beep." The paging format will be specific to each organization and thus requires modification. The pager configuration uses a specific email address that, when emailed with a PIN and message, contacts the pager company and arranges to send the message to the alpha pager corresponding to the PIN. Some administrators may enjoy early morning pages, but I found it as just as useful to send an email, then shut off the network interface, so that the system is "locked down" until the administrator gets to the system console.

- For `syslog` configuration, enter the logging priority or press `<Enter>` for the default of `auth.crit.` (Enter `man logger` for more info).

- There are no options for the Shutdown Network Interface Option. It simply waits five seconds to allow email, etc. to be delivered, then shuts down all network interfaces except loopback. Keeping loopback up allows local networking services (such as X-Windows if you are not using Unix domain sockets) to function normally, but blocks all remote access.

- Likewise, there are no options for the System Shutdown Option. It waits five seconds, then executes `init 0`, which will shut down the system.

Note that the options are executed in numerical order, so that you can send email, then page the administrator, then post to `syslog`, then shut down the network, then shut down the system itself.

A shell command that is displayed prior to the actual patching process is constructed from the selected options. At this point you are given the option of editing the script. The editor to be run is defined by the `EDITOR` environment variable; the default is `vi`. When the command is as you want it, type "0" to patch the binary. The shell command is run by `sh`, which thus requires that you use the appropriate semantics.

The `patcher` program embeds and encrypts the shell script directly in the binary. When complete, your custom UPDATE binary will be in the current directory, and can be installed in the startup binary directories by copying it to `/shin` or `/usr/sbin`, depending on where they are normally installed. If you already have an UPDATE program, rename it to "UPDATE0."

Next, I ensure that this program is well-hidden. The permissions and ownerships should be the same as other programs in the install directory. Set the timestamp back by entering:

```
chown bin update
chgrp bin update
touch -r wall update
```

I use the `wall` program as a reference timestamp, but you should look in the install directory for a program with an innocuous time stamp.

### Testing

To initially test UPDATE, simply run it as root-equivalent, using the same shell that is used during the boot process (usually `/bin/sh`, `/bin/ksh`, or `/bin/bash`). UPDATE will immediately put itself in the background and start doing its magic. To make it run on system startup, add to your `/etc/rc*` files. UPDATE ignores all signals, so to turn it off you must execute a `kill -9`.

It is also wise to rename UPDATE. If attackers identify that a program named "UPDATE" is running, they are likely to do something undesirable (especially if they have root access). Fortunately, there is no reason the program needs to be named "UPDATE"; any innocuous name will do. UPDATE can even be started via `cron` (using the `-once` flag) or manually.

After completing initial testing and making any changes dictated by the test results, I suggest performing a final test of UPDATE before you rely on it. Run `tcpdump`, `snoop`, `netsnoop`, or another command to sniff the network. Within three minutes you should receive email (provided, of course, that you have configured the email option) notifying you that the system has a security violation. If the sniffer is running when UPDATE is started, you will get an immediate response, so testing will in this case be faster.

## Command Line Options

Various command line options to UPDATE are available. The `-once` option, for example, causes UPDATE to run one time, then exit, something that is valuable for testing purposes. UPDATE also allows up to two numeric arguments, namely the time delay in seconds between `sync` calls and sniffer detection, respectively. Defaults (30 seconds for `sync`, 180 seconds for sniffer detection) are used whenever arguments are omitted or invalid. For users with a large RAID system or other method for flushing the buffers, `sync` calls can be effectively disabled by setting `sync` to a suitably large number. A setting of 32000000 will, for example, create a time delay between `syncs` of more than a year. You can also enter "`UPDATE 86400 300`" to run `sync` every day (86400 seconds) and sniffer detection every 300 seconds.

## Conclusion

UPDATE has become the standard sniffer detector software in the organization for which I previously worked. It is an important part of an integrated security program that includes firewall deployment, intrusion detection, vulnerability scanning and security awareness training. UPDATE has successfully detected and deflected many attacks in which sniffers were installed without authorization. Support from the user community has been uniformly positive and helpful — many of the enhancements and improvements have in fact been driven by dedicated system administrators. Any suggestions or questions from readers are also welcome. I hope you find UPDATE as useful as many others have.

## Acknowledgements

Many people have contributed to making UPDATE successful, but the following individuals[5] are worthy of special mention: David Curry (IBM) (for writing the original ifstatus program), Neal Mackanic for suggestions on improvement, Dave Temple for the idea of shutting down the network interface, Ken Underhill for user interface suggestions, Dave Martini for allowing me access to some of his machines to support more interfaces under Solaris, Dave Williams for access to a Solaris x86 system, Jimmy Guse for information on FreeBSD, and all beta testers.

[5] I apologize to anyone whose name I have omitted. If you email me, I'll include you in all subsequent publications and reports about UPDATE.